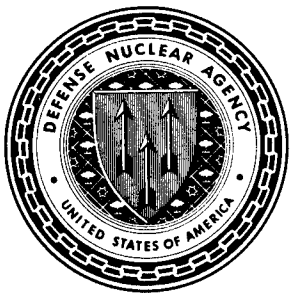
Defense Nuclear Agency
Alexandria, VA 22310-3398



DNA-TR-94-64

Microcomputer Visualization of Nuclear Cloud Models

Ernest T. Wright
Virtual Image Labs Inc
1738 Elton Road # 307
Silver Spring, MD 20903

December 1994

Technical Report

CONTRACT No. DNA 001-92-C-0173

Destroy this report when it is no longer needed. Do not return to sender.

PLEASE NOTIFY THE DEFENSE NUCLEAR AGENCY, ATTN: CSTI, 6801 TELEGRAPH ROAD, ALEXANDRIA, VA 22310-3398, IF YOUR ADDRESS IS INCORRECT, IF YOU WISH IT DELETED FROM THE DISTRIBUTION LIST, OR IF THE ADDRESSEE IS NO LONGER EMPLOYED BY YOUR ORGANIZATION.

DNA PRINTING PLANT

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	941201	Technical 921001 – 940331

4. TITLE AND SUBTITLE

Microcomputer Visualization of Nuclear Cloud Models

5. FUNDING NUMBERS

C - DNA 001-92-C-0173
PE - 62715H
PR - AC
TA - CE
WU - DH328130

6. AUTHOR(S)

Ernest T. Wright

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Virtual Image Labs Inc
1738 Elton Road # 307
Silver Spring, MD 20903

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Nuclear Agency
6801 Telegraph Road
Alexandria, VA 22310-3398
SPWE/Byers

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

DNA-TR-94-64

11. SUPPLEMENTARY NOTES

This work was sponsored by the Defense Nuclear Agency under RDT&E RMC Code B4662D AC CE 00008 7010A AC 25904D.

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report describes ICCanvas, a computer program which creates and displays isosurfaces on 3D scalar fields. ICCanvas runs under Microsoft Windows 3.1 on computers with 4 MB or more of RAM. Its three program files total less than 150 KB, and its fully documented custom file format for scalar fields is highly compressed. The ICCanvas distribution disk includes C language source code for creating IC format data files.

ICCanvas was developed for personal computer visualization of 3D density data created by the TASS and DICE-MAZ nuclear cloud hydrodynamic codes. Its scalar field data model is sufficiently general that it may also be used to view the results of 3D medical imaging (e.g., CT and MRI), geological surveys, or the tabulated values of almost any function of three variables.

ICCanvas gives users interactive control over the orientation of the view and the choice between wireframe, faceted and Gouraud-shaded rendering. The program's display can be saved in Windows BMP, PC Paintbrush PCX, and Macintosh PICT formats, and the geometry of the surfaces it creates can be saved as AutoCAD DXF and Wavefront OBJ files. The program includes extensive on-line help.

14. SUBJECT TERMS		15. NUMBER OF PAGES
Windows	3D	42
Nuclear Cloud	Scalar Field	16. PRICE CODE
Nuclear Weapons Effects	Visualization	

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR

CLASSIFIED BY:

N/A since Unclassified.

DECLASSIFY ON:
N/A since Unclassified.

TABLE OF CONTENTS

FIGURES

SECTION 1
INTRODUCTION

> Given for one instant an intelligence which could comprehend all the forces by which nature is animated and the respective positions of the beings which compose it, if moreover this intelligence were vast enough to submit these data to analysis, it would embrace in the same formula both the movements of the largest bodies in the universe and those of the lightest atom; to it nothing would be uncertain, and the future as the past would be present to its eyes.
>
> **—Pierre Simon de Laplace**
> *Analytic Theory of Probabilities*, 1820

Modern physical theory no longer permits us, even in principle, to be so sanguine about the certainty of the future, but we continue to rely on numerical models of events that have not yet happened, models built on what Laplace called a calculus of common sense, to allay our ignorance. Computers have lately acquired the power to animate these models at a level of detail that threatens to exceed our finite comprehension only slightly less than does the real world. Fortunately, France also gave us Descartes.

1.1 BACKGROUND.

The Defense Nuclear Agency has sponsored the compilation of a library of computer-generated nuclear clouds that allows military planners to accurately characterize the atmospheric environment near the site of a nuclear burst. The library comprises two- and three-dimensional arrays of densities, along with related flow field and temperature data, calculated for different weapon yields, heights of burst, times after burst, and particle types and sizes. Both the library itself and the hydrodynamic codes that produced it reside on a Cray supercomputer at the Los Alamos National Laboratory.

Visualization, the creation of images from numerical data, is an important part of the research and analysis that both inform and follow from the calculation of nuclear cloud dynamics. But to visualize data as prolific as the 3D arrays of density values developed by the hydrocodes, the only option has been the use of high-end computers running expensive data analysis software—Stardent's Advanced Visualization System (AVS) is an example. Most people with a valid interest in using the library don't have ready access to such systems.

This report describes a computer program called ICCanvas ("Isosurface Constructor Canvas"), an interactive 3D visualization tool that runs in the Microsoft Windows environment. ICCanvas allows personal computer users to view level surfaces on structured 3D data. Typical sources of data include the outputs of 3D TASS and DICE-MAZ, but the program's data model is sufficiently general that it may also be used to

visualize data from 3D medical imaging (e.g., CT and MRI), geological surveys, or the tabulated values of almost any function of three variables.

ICCanvas supports a custom file format for 3D data arrays that can cut the size of a cloud file by two orders of magnitude or more. The program can also export the images and surfaces it creates, a handoff capability that can link the cloud library to off-the-shelf CAD and 3D rendering applications.

1.2 REPORT OBJECTIVES.

This report is a conceptual description of the algorithms employed by ICCanvas and of some of the ways the program can be used. Section 2 discusses in some detail the approach to surface construction at the heart of the program, along with the viewing transformations and rendering methods used to create surface images. Section 3 includes a tour of the program from the user's point of view and an introduction to the use of other 3D rendering applications with ICCanvas, but the bulk of this section is devoted to documenting the IC file format. The Appendix contains an enumeration of each of the 256 entries in the cube case array explained in 2.1.2. The triangles for each cube were produced by actually submitting the cube to ICCanvas as a (very small) density array and then rendering the output. This output serves as a straightforward, albeit tedious, verification of the code.

Many aspects of our work on ICCanvas, while accounting for a substantial amount of the time spent on the program's development, are not treated at length in this report, primarily because they can't be discussed at a level of generality that's accessible (or of interest) to most readers. These include the mechanics of writing a Windows application, strategies for handling large amounts of data in a 16-bit environment, the subtleties of creating native-format image and geometry files, and the broad field of 3D rendering. The interested reader will find this information in a number of readily available sources.

SECTION 2
ALGORITHMS

2.1 SURFACE CONSTRUCTION.

A *scalar field* is a 3D array of values of a function $f(x,y,z)$ tabulated at regular intervals of x, y and z. In IC, the scalar field values usually represent densities, but they may be the values of other functions of three variables—temperature, field strength, or probability, for example. IC imbeds the scalar field in a regular grid consisting of edges that join neighboring scalar field values. It then constructs a triangle mesh approximation of a *level surface* or *isosurface* on the scalar field. A level surface consists of all points (x,y,z) for which $f(x,y,z) = c$ for a constant c, called the *level*.

The triangle mesh approximation, called simply the *surface* in what follows, is represented by an array `VERT[]` of vertex coordinates and an array `FACE[]` of triangles. Each triangle in `FACE[]` is defined by three indexes into the `VERT[]` array, and each vertex is defined by three space coordinates. The final sizes of the `VERT[]` and `FACE[]` arrays are unknown until surface construction is complete, so vertex and face information is written to temporary files, which are read back in and deleted when the build is finished.

The surface finding approach described in 2.1.1 and 2.1.2 most closely resembles Marching Cubes, a method described by Lorensen and Cline (1987). Similar methods have been developed by Wyvill et al. (1986) and Bloomenthal (1988). See Hall (1990) for a summary of all three methods. Additional remarks on Marching Cubes can be found in Dürst (1988), Wilhelms and Van Gelder (1990) and Foley et al. (1990).

2.1.1 Finding Vertices.

The routine `icFindVertices()` finds surface vertices on grid edges for which

```
(f[n] ≥ level) XOR (f[n+1] ≥ level) = TRUE
```
(2.1)

where `[n]` is an array index `[i,j,k]` into the scalar field `f`, `[n+1]` represents one of `[i+1,j,k]`, `[i,j+1,k]`, `[i,j,k+1]`, and `XOR` is the logical exclusive-or operator. This is just a formal way of saying that a surface vertex is found on every grid edge whose endpoints have values on opposite sides of the level.

For each edge satisfying Eq. (2.1), two of the three coordinates of the surface vertex are fixed by the position of the edge, and the third is found by linear interpolation:

$$u_{frac} = 1 + 254 * (f[n+1] - level)/(f[n+1] - f[n]) \qquad (2.2)$$

where `u` is a coordinate in the array index coordinate system. (Array index coordinates are 16-bit fixed-point numbers whose integer part is `n` and whose fractional part is a fraction of the distance from `f[n]` to `f[n+1]`.) Internally, the scalar field and level are

expressed as logarithms (see 3.2.3), so that the interpolation is actually linear in log space. Note too that the calculation is performed using integer arithmetic, and that the constants in Eq. (2.2) prevent the surface vertex from lying exactly at the position of a scalar field value.

The coordinates of each surface vertex are written to the vertex temporary file as they are found, and an ordinal for the vertex is associated with the grid edge so that it can later be retrieved by the surface triangle routine `icFindFaces()` described in 2.1.2.

2.1.2 Finding Faces.

The routine `icFindFaces()` finds surface triangles by sequentially examining *cubes* in the grid. A cube consists of eight scalar field values, one at each corner, connected by twelve grid edges. The relationship of the corner values to the level is written into a single 8-bit integer, called a cube case number, by

```
cube[n] = (in[i,j,k]         LSHIFT 0)
        + (in[i+1,j,k]       LSHIFT 1)
        + (in[i+1,j+1,k]     LSHIFT 2)
        + (in[i,j+1,k]       LSHIFT 3)
        + (in[i,j,k+1]       LSHIFT 4)
        + (in[i+1,j,k+1]     LSHIFT 5)
        + (in[i+1,j+1,k+1]   LSHIFT 6)
        + (in[i,j+1,k+1]     LSHIFT 7)                    (2.3)
```

where `in[n]` = 1 when `f[n]` ≥ `level` and 0 otherwise, and `LSHIFT` is the bitwise left shift operator. The numbering of cube corners implied by Eq. (2.3), in which the binary state `in[n]` for each cube corner is mapped onto one of the eight bits of the number `cube[n]`, is illustrated in the Key to the cube case array (Figure A-1) given in the appendix. Note that there are 256 cubes because there are $2^8 = 256$ possible values for `cube[n]`.

`cube[n]` forms an index in the range [0, 255] into an array `vcase[]` containing triangle descriptions for each variety of cube. Each triangle in `vcase[]` is defined in terms of the cube edges, numbered from 0 to 11, on which each of its three vertices lie. For the cube at `n`, each of the cube edges in `vcase[cube[n]]` corresponds to a grid edge with endpoints `f[n]`, `f[n+1]` that satisfy Eq. (2.1). Recall from 2.1.1 that vertex ordinals have already been associated with these edges. Each surface triangle may therefore be defined by three vertex ordinals collected from the appropriate grid edges. The vertex ordinals for each surface triangle are written to the face temporary file.

2.1.3 Surface Compaction.

Surfaces built as described in 2.1.1 and 2.1.2 tend to contain a significant number of small triangles that burden the graphics routines without contributing much to the shape of the surface. Some of these triangles may be long, thin slivers that are ill-conditioned or nearly

degenerate for some rendering algorithms. IC implements a solution called Compact Cubes described by Moore and Warren (1991, 1992).

In the pseudocode that follows, a *gridpoint* **g** is just the position of a scalar field value. A surface vertex is called a *satellite* of **g** if it is closer to *g* than to any other gridpoint.

```
for each surface triangle T
    if the vertices of T are satellites of different gridpoints
        then produce a new triangle connecting the gridpoints
        else T collapses to a vertex or an edge, so ignore it
end for
for each gridpoint g of the new triangle mesh
    displace g to the average position of its satellites
end for
```

The initial use of array index coordinates to define vertices (see 2.1.1) reduces the task of determining satellite relationships to a trivial rounding of the coordinates and simplifies the production of new triangles.

2.1.4 Completing the Surface Definition.

Once all of the vertices and faces have been defined, they are read back in from the temporary files, which are destroyed. Vertex coordinates are scaled so that the space of the scalar field just fits inside a unit cube. The conversion from array index coordinates to a floating-point system with values in [0, 1] (in the context of viewing discussed in 2.2, the *world* coordinate system) simplifies the graphics routines while restoring the aspect of the surface—the sampling intervals along each of *x*, *y* and *z* for the scalar field are uniform, but in general they are not the same for different axes.

A normal for each face is calculated as a cross product of two vectors lying along edges of the face. For vectors **p** and **q** defined as

$$\mathbf{p} = (x_2 - x_1,\ y_2 - y_1,\ z_2 - z_1),$$
$$\mathbf{q} = (x_1 - x_3,\ y_1 - y_3,\ z_1 - z_3)$$

where the *x*, *y* and *z* are coordinates of the three face vertices, the normal **n** is the cross product $\mathbf{p} \times \mathbf{q}$ normalized to unit length. Note that two directions for **n**, symmetric with respect to sign, are possible, and that the choice can depend on the order in which the vertices are listed. All of the triangles listed in the `vcase[]` array therefore specify vertices in clockwise order as seen from outside the surface, so that $\mathbf{p} \times \mathbf{q}$ points outward. When a single winding order has been adopted in this way for all vertices, the polygons are sometimes said to be *oriented*.

2.2 THE VIEWING TRANSFORMATION.

The coordinates of the VERT[] array are specified in the world coordinate system. In order to create 2D images that allow the user to see a surface from arbitrary points of view, IC creates a view coordinate system and a transformation from world to view coordinates that projects surface triangles onto a viewplane. The user controls the view through settings of the azimuth θ, latitude ϕ and center point **r**. Azimuth and latitude determine the orientation of the viewplane normal **n** and up unit vector **v** in the following way:

$$n_x = -\cos\phi\cos\theta$$
$$n_y = -\cos\phi\sin\theta$$
$$n_z = -\sin\phi$$
$$\mathbf{v} = \frac{\mathbf{z}-(\mathbf{z}\cdot\mathbf{n})\mathbf{n}}{\left|\mathbf{z}-(\mathbf{z}\cdot\mathbf{n})\mathbf{n}\right|}, \quad \mathbf{z}\equiv(0,0,1)$$

Note that **v** is just the projection of the z-axis perpendicular to **n**. This construction is degenerate for **v** when the user wants to look straight down or straight up (**n** parallel to **z**), and for that case it is replaced by

$$n_x = 0$$
$$n_y = 0$$
$$n_z = -\mathrm{sign}\phi$$
$$v_x = -\mathrm{sign}\phi\cos\theta$$
$$v_y = -\mathrm{sign}\phi\sin\theta$$
$$v_z = 0$$

where $\mathrm{sign}\phi \in \{1,-1\}$. For all cases, the rightward unit vector **u** must be perpendicular to both **n** and **v**, a condition satisfied by the vector cross product

$$\mathbf{u} = \mathbf{n}\times\mathbf{v}$$

The center point **r** becomes the origin of the left-handed view coordinate system with axis directions **u**, **v** and **n**. Note that **r** and **n** define the viewplane and **v** determines the orientation of the rectangular window (see Figure 3-1). By defining an intermediate quantity **r**′ as

$$\mathbf{r}' = (-\mathbf{r}\cdot\mathbf{u}, -\mathbf{r}\cdot\mathbf{v}, -\mathbf{r}\cdot\mathbf{n})$$

the world to view transformation $\mathbf{T}_{WV}$ may be captured by the 4 × 4 matrix

$$\mathbf{T}_{WV} = \begin{bmatrix} u_x & v_x & n_x & -n_x/e \\ u_y & v_y & n_y & -n_y/e \\ u_z & v_z & n_z & -n_z/e \\ r'_x & r'_y & r'_z & 1-r'_z/e \end{bmatrix}$$

The scalar e is a displacement along **n**, in view coordinates, from the viewplane to a point called the *eye*. This distance, which in IC is fixed, controls the amount of perspective foreshortening in the image.

Readers might recognize the viewplane normal **n**, origin **r** (also called the view reference point or center of projection) and other parts of this development as variations on the synthetic camera model, informative descriptions of which are available from almost any good 3D graphics text (see, for example, the textbook by Hill (1990)).

A further scale and translation map from view coordinates to screen pixels. This transformation takes into account the user-defined zoom scale factor s_{zoom} and the aspect, or relative width and height a_x, a_y, of the display's pixels. The view is fit into a rectangular display window so that a distance of $1/s_{zoom}$ just fits the smaller of the two window dimensions d_x, d_y.

$$s_u = \begin{cases} s_{zoom} d_x, & d_x a_x < d_y a_y \\ (a_y/a_x) s_{zoom} d_y, & d_x a_x \geq d_y a_y \end{cases}$$

$$s_v = \begin{cases} (-a_x/a_y) s_{zoom} d_x, & d_x a_x < d_y a_y \\ -s_{zoom} d_y, & d_x a_x \geq d_y a_y \end{cases}$$

$$\mathbf{T}_{screen} = \begin{bmatrix} s_u & 0 & 0 & 0 \\ 0 & s_v & 0 & 0 \\ 0 & 0 & 1 & 0 \\ d_x/2 & d_y/2 & 0 & 1 \end{bmatrix}$$

In other words, view coordinates are scaled horizontally by s_u and vertically by s_v, and the pixel origin is moved to the center of the window. Note that s_v is always negative, which has the effect of performing a vertical flip. By default in Microsoft Windows, the pixel at (0, 0) is in the upper left corner of the client area of the window and the vertical coordinate increases downward.

The complete viewing transformation **T** is then

$$\mathbf{T} = \mathbf{T}_{WV}\, \mathbf{T}_{screen} .$$

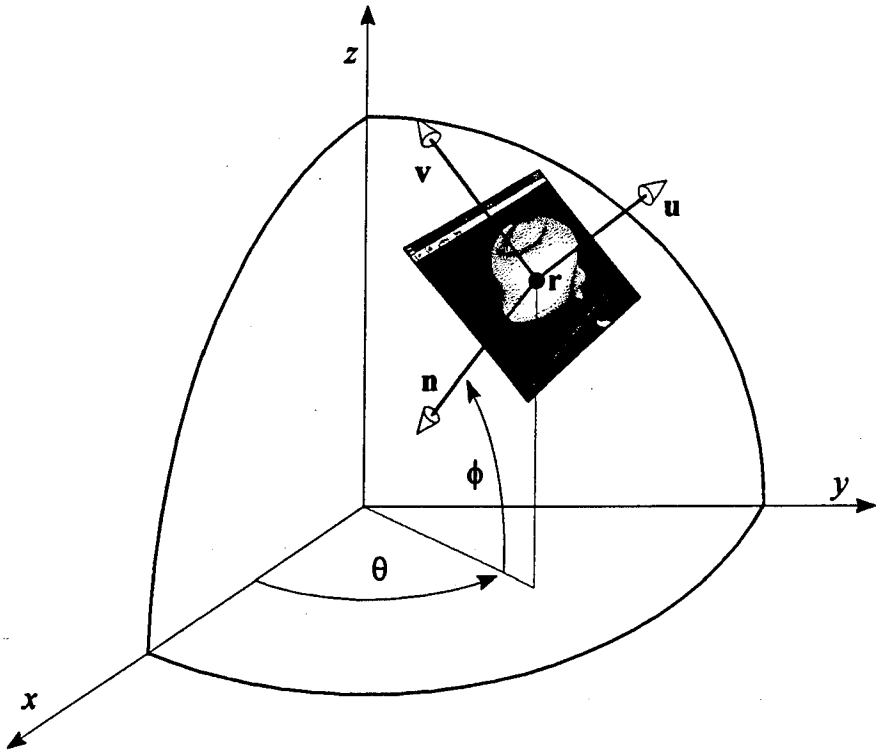


Figure 2-1. Elements of the viewing system. The 3D surface geometry in *xyz* is not shown.

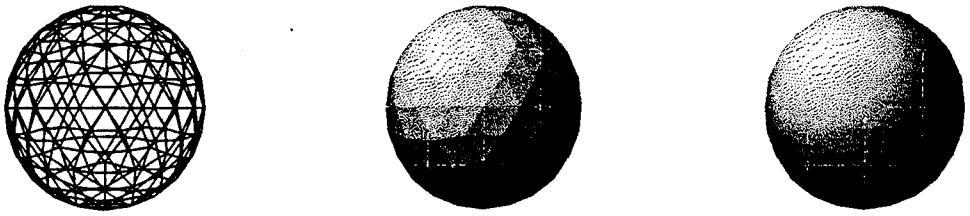
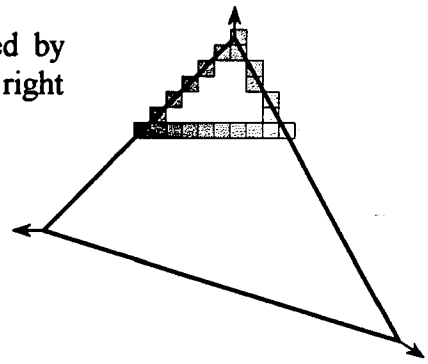


Figure 2-2. A tessellated sphere rendered by three different methods. The triangle at right illustrates intensity interpolation.



A point **p** in VERT[], extended to include a homogeneous coordinate $w = 1$,

$$\mathbf{p} = (p_x, p_y, p_z, 1)$$

is transformed by

$$\mathbf{q} = \mathbf{pT}$$
$$\mathbf{p}' = (q_x / q_w,\ q_y / q_w,\ q_z / q_w)$$

2.3 RENDERING AND SCAN CONVERSION.

One of the advantages of the surface finding method described in 2.1 is that it produces a surface defined entirely by a list of triangles. Planar convex polygons are among the simplest and best understood objects in 3D graphics, and a large number of methods is available for drawing them on the display. IC implements three of them, and for the user the choice between them is a tradeoff involving image quality, information content and rendering speed (see Figure 3-2).

The simplest surface representation is a wireframe drawing, produced by drawing the edges of each surface triangle. Wireframes are fast. They make the polygonal geometry of the surface visually explicit, and since they're transparent, they can show all sides of the surface simultaneously. They cannot, however, provide many of the important 3D visual cues that users take from a surface drawn more realistically as a shaded solid object. For this the user may choose between facet rendering, which remains close to the polyhedral surface definition, and more computationally expensive smooth rendering, which treats the polygon mesh as an approximation to a continuous surface.

2.3.1 Hidden Surface Removal.

Both facet and smooth rendering require the removal of hidden parts of the surface—those parts of the surface that are behind other parts and therefore obscured from view. Because of the nature of the surfaces produced by the program, IC gets good results using very straightforward techniques called *back face culling* and *depth-order traversal*.

Back faces are triangles facing away from the camera, meaning roughly that their normals form an angle with the direction of view of more than 90°. In most cases, IC surfaces are closed polyhedra for which back faces always represent the enclosed and therefore non-visible interior side of the surface. (The exception is a surface truncated at the edge of the data.) Back faces are removed from consideration early in the rendering process, which on average cuts in half the number of triangles to be rendered.

Depth-order traversal sorts triangles from far to near and then renders them in that order. When possible ambiguities after sorting are ignored, this approach is also known as the painter's algorithm, a reference to the way artists sometimes paint closer objects on top of

more distant ones. In the general case, where polygons may have overlapping extents or even interpenetrate, the painter's algorithm will produce incorrect results. The triangles in an IC surface, however, are built on a regular grid, which restricts their interactions to cases for which the painter's algorithm nearly always succeeds.

2.3.2 Lighting and Shading

Lighting and shading are distinct aspects of rendering, but in IC they are both modeled simply enough that they may be treated together. IC creates surface images in varying shades of gray by putting a point light source at the position of the camera and then calculating the intensity of the diffuse reflected light at different points on the surface. IC's point light source is just a vector giving the direction *opposite* to the direction of illumination. The diffuse reflected intensity at a point on the surface is then the cosine of the angle of incidence, which is found as the dot product of the light vector and the surface normal at that point.

Facet and smooth rendering differ in the choice of surface normals and in the number of intensity calculations. Facet rendering uses the face normals developed in 2.1.4 to assign a shade of gray to an entire triangle. This constant shading accounts for the surface's faceted appearance. Smooth rendering uses a technique called Gouraud interpolated shading (Gouraud 1971).

IC develops surface normals at the vertices of triangles by summing the components of the face normals for faces sharing each vertex, and for each triangle an intensity is calculated for its three vertices using the dot product relation. Gouraud shading finds intensities for points on triangle edges by linear interpolation between the endpoint vertices. An intensity for any interior point of the triangle may then be found by interpolation between the two edges that cross that point's scanline, the horizontal row of pixels in which the transformed point lies.

For facet rendering (and wireframes), the job of setting individual pixel values on the display for each triangle, called *scan conversion*, is performed by low-level graphics routines in the Windows video driver for the display hardware. The IC DLL passes the pixel coordinates of the triangle vertices to the Windows function `Polygon()` (`Polyline()` for wireframes). Gouraud shading, however, is an integral part of scan converting a triangle, and IC must therefore perform its own scan conversion. The scan conversion routines in IC closely follow code by Heckbert (1990).

SECTION 3
RESULTS

3.1 AN INTERACTIVE PC-BASED PROGRAM.

Over the contract period, ICCanvas has evolved from a DOS program with limited viewing and display capabilities, modest mouse support, a text-based interface and substantial memory constraints into a full-fledged 3D visualization tool for Windows with online help, complete viewing controls, Gouraud interpolated shading, and the ability to export both images and 3D geometry. The paragraphs that follow briefly describe the operation of the program in its current form from the user's point of view. See Figure 3-1 for an idea of what the user actually sees.

The ICCanvas program package consists of the program file iccanvas.exe, the dynamic-link library iclib.dll, which contains functions implementing the algorithms of Section 2, and the online help file ichelp.hlp. The package is installed by copying these three files to the user's hard drive and performing the usual Windows procedure for making the program's icon visible and selectable from the Windows Program Manager.

To begin an ICCanvas session, the user picks the Open menu option in the File menu to open an IC format data file containing a 3D array of densities. The program examines this file and displays an approximate distribution of the densities it contains using the Level dialog. By default the level is set to the median value in this distribution, but it may be changed to any value.

Once the data file is chosen and the level set, ICCanvas reads the data and constructs a surface. The progress of the build is displayed, and during most of that time (typically 5 to 10 seconds) the user is given the option of cancelling the build. Assuming it is allowed to finish, the program then draws the surface on the display.

To view the surface from different angles, the user can click and hold the left mouse button anywhere in the display, and by moving the mouse change the orientation of the view. A picture of a square representing the ground plane with a vertical axis attached at one corner (the axis image) is drawn on top of the surface while the left mouse button is held down. Moving the mouse changes the orientation of the axis image, and when the left mouse button is released the surface is redrawn at the new viewing angle.

The user is given greater control over the view through a Camera dialog and Center menu option, both available from the Settings menu. The Camera dialog allows the user to enter the viewing angles, called azimuth and latitude, in degrees. The angle edit fields are linked to a miniature axis image, and when either the text or the image is manipulated, the other is updated to reflect the change. Another control, called Zoom, allows the user to magnify or shrink the image in the display. The Camera dialog also offers six predefined views (Top, Front, etc.) to simplify the specification of axis-aligned orientations.

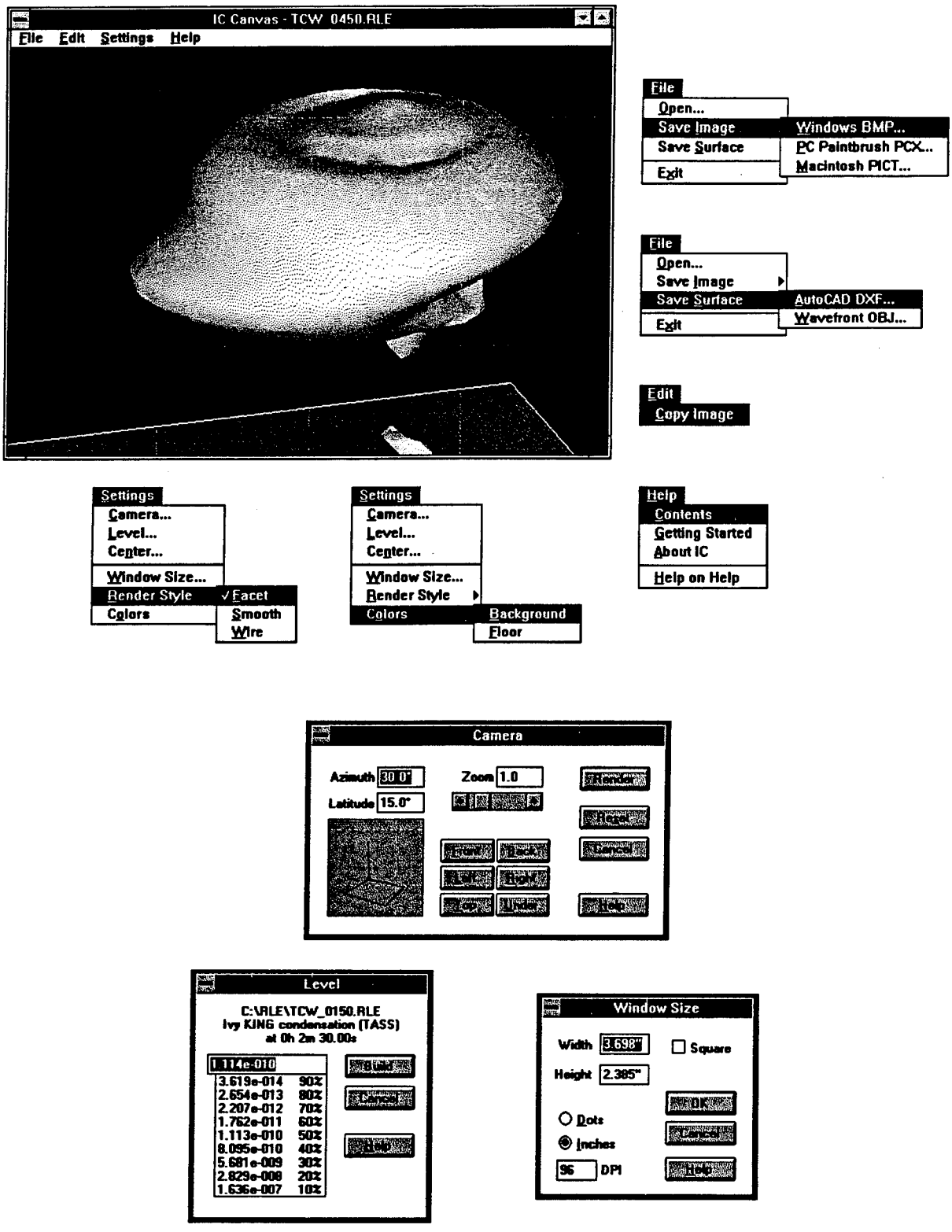


Figure 3-1. Elements of the ICCanvas user interface.

The Center menu option is used to pick a new center point for the display. The mouse cursor is changed to a crosshair, and a small window is opened to report the position of the cursor in 3D object coordinates as it moves. While centering, the mouse cursor is actually moving in 3D along the plane of the display, and this can be useful, within limits, for locating specific features on a surface.

To build a surface from the same data but with a different level, the user can call up the Level dialog from the Settings menu. The table of levels displayed in the Level dialog tries to give some indication of the distribution of densities in the data. Each level is paired with a percentage that tells how many of the *non-zero* data points will be enclosed by a surface at that level. In this respect, ICCanvas assumes that the data represents a mass of some arbitrary shape sitting in a rectangular void. The statistics in the list attempt to be relevant to the mass and not to the void.

The surface can be drawn, or rendered, in one of three ways controlled by the Render Style menu options in the Settings menu. By default, surfaces are rendered in the Facet style, which uses constant shading to draw each surface triangle. The results are fast and solid-looking but somewhat rough-hewn. Smooth rendering uses Gouraud interpolated shading, which looks better but takes longer. The very fast Wire option completely reveals the triangle mesh.

ICCanvas also allows the user to transfer images and surface geometry to files or to other programs through the Save Image, Save Surface, and Copy Image menu options. Images can be saved to disk as Windows BMPs, PC Paintbrush PCX files or Macintosh PICTs. The display can also be copied to the Windows clipboard, to be retrieved by another program currently running. The Save Surface options write the 3D geometry of a surface in either AutoCAD DXF or Wavefront OBJ format. Because of its complexity, the Print option isn't available in ICCanvas, but the ability to export both images and geometry gives the user the choice of any print-capable off-the-shelf software for producing ICCanvas hard copy.

For better support of image export, the user can control the colors of the background and the floor as well as the precise dimensions of the display. The Window Size dialog reports the size of the display (strictly speaking, the size of the windows's *client area*) in either pixels (picture dots) or inches.

3.2 DATA COMPRESSION AND THE IC FILE FORMAT.

On larger computer systems, storage has historically been cheap and programming effort expensive. This economy, combined with the widespread use of FORTRAN in the simulation community, has led to the storage of 3D hydrocode results in collosal text files measured in gigabytes. In order for IC to be practical as a PC-based application, it was recognized very early that a faster and much smaller file format for density data was needed.

3.2.1 Text vs. Binary

The performance penalty in reading a large text file arises partly from the file's size, but it is mostly a result of the representation of numbers as strings of characters. When reading numbers as text, a program must feel its way one character at a time, first to parse the character stream into individual numbers by looking for delimiters (commas and spaces, for example), and then to convert the digit string (really a polynomial in the number base, 10) into a base 2 bit string understood by the machine. Binary files that store numbers in machine-readable form eliminate the need for both parsing and conversion; the increase in reading speed over text files is often two orders of magnitude. Binary files are usually smaller, too.

To better appreciate the impact this has on program performance, as well as the influence of user expectations, consider that each frame—1/30 second—of broadcast television, if stored as discrete samples of the NTSC waveform, would require well over a megabyte of text and take on the order of tens of seconds for a computer to read from a hard disk. No one would watch a TV system with a refresh rate of 0.05 Hz, and the inertia of large text files similarly mitigates against the usefulness of 3D density arrays.

Since the output of 3D TASS and DICE-MAZ, for example, isn't meant to be read directly by humans, the primary benefit of text output for these codes is that it is easily transported between machines that may have different binary number representations. This is especially true for files produced on supercomputers, whose idiosyncratic internal number formats are shared by very few machines. With a little effort, however, it is almost always possible to write binary files on one platform that can be read on another, and with its enormous installed base, the PC platform is a reasonable target for such effort. Figure 3-2 provides an example of how this is done.

3.2.2 Run-Length Encoding.

Run-length encoding (RLE) eliminates redundancy in a data array by replacing runs of equal values with a count followed by a single instance of the value. In what is sometimes called *adaptive* RLE, the count is also a code that distinguishes between repeat runs and literal runs, which eliminates the need for a count of 1 before each data value not part of a repeat run. IC uses the following interpretation of its RLE codes (`n`):

$n \geq 0$	a run of $n + 1$ data values follows
$n < 0$	repeat the next value $-n + 1$ times

The sequence

```
1 2 3 3 3 4 4 4
```

is encoded as

```
/*
=========================================================================
make_float()

Convert a float to an IEEE bit string.

INPUTS
   x        floating-point value

CALLS
   fabs()   math.h, floating-point absolute value
   frexp()  math.h, split FP number into exponent and mantissa

RESULTS
   The value of x is written to a byte array as an arrangement of bits
   representing an IEEE 4-byte float in Intel byte order, and the byte
   array is returned.

+-----------------------------------------------------------------------+
|  0        1        2        3               Bit pattern for the     |
|  76543210 76543210 76543210 76543210        IEEE 4-byte float in    |
|  mmmmmmmm mmmmmmmm eeeeeeee seeeeeee        Intel byte order.       |
+-----------------------------------------------------------------------+
|  sign     exp      mant     value                                    |
|  ------------------------------                                      |
|     0       0         0       0.0                                    |
|     0     127         0       1.0           IEEE representations     |
|   any     255         0       INF           of some values of        |
|   any     255     not 0       NAN           interest.                |
+-----------------------------------------------------------------------+
=========================================================================
*/

static char *make_float( float x )
{
   static char c[ 4 ];
   int e, s;
   unsigned long m;
   double d;

   d = fabs( frexp(( double ) x, &e ));
   m = ( unsigned long )( d * ( 1L << 24 )) & 0x007FFFFF;
   s = ( x < 0.0 );
   e += 126;

   c[ 0 ] = m & 0xFF;
   c[ 1 ] = ( m >> 8 ) & 0xFF;
   c[ 2 ] = (( m >> 16 ) & 0x7F ) | (( e & 1 ) << 7 );
   c[ 3 ] = (( e >> 1 ) & 0x7F ) | ( s << 7 );
   return c;
}
```

Figure 3-2. Example source code for the creation of an Intel/IEEE 4-byte float on any machine.

```
1 1 2 -2 3 -2 4.
```

Evidently, the amount of compression varies with the number of values that are part of repeat runs. In the average case for typical IC data, there are long runs of 0's that account for most of the extra space squeezed out of a file by RLE. In the worst case, which contains no repeat runs longer than 2, an optimal implementation of RLE will add no more than 1 to the length of a data array. Optimal RLE doesn't try to encode repeat runs of length 2. Consider the sequence

```
1 2 3 3 4.
```

Treating the 3's as a repeat run yields

```
1 1 2 -1 3 0 4
```

whereas encoding the whole sequence as a literal run leads to

```
4 1 2 3 3 4.
```

3.2.3 A 16-Bit Real Number Format.

In IC, floating-point density values x are scaled into 2-byte integers by the C-language expression

```
x <= 1e-16 ? 0 : 2048 * log10( x ) + 32768
```

In terms of precision and range, this is equivalent to using 5-place common logarithms for values between 10^{-16} and 10^{16}. There are a number of advantages in using this format for densities in memory as well as in data files, some of which are listed here.

1. 2-byte integers are smaller than the 4-byte and 8-byte binary formats available for floating point numbers. This has a significant effect on file size, but it more importantly affects the size of density data in memory. In general, segmented memory addressing under MS-DOS places an upper limit of 64K on the amount of memory that can be addressed by a single pointer. Within this limit, a 2D array of density values can be as large as 181 × 181 with the 2-byte format, while an array of 4-byte floats can be no larger than 128 × 128.

2. Integer arithmetic is faster than floating-point. (The only actual arithmetic operations performed on density values are comparison, for which the format is numerically irrelevant, and interpolation, for which the use of log scaling has the beneficial side effect of creating smoother surfaces on data with large gradients.)

3. Because of the way surface points are inferred from the density data, 16 bits is just the right amount of precision for this data (assuming, fairly safely, that the data varies over a significant fraction of the number format's range).

4. RLE is more difficult to implement efficiently when the codes and data values are of different numeric types, especially in languages other than assembler and C.

5. Using the same format in memory and on disk eliminates the need to perform a conversion when moving from one to the other.

3.2.4 Other Compression Methods.

Other compression schemes were considered. At least one PC-based application for rendering landscapes from U.S. Geological Survey data stores arrays of 16-bit elevations in runs of 8-bit deltas, or changes in elevation. This works because most elevation gradients are sufficiently small that the change between sample points is usually less than 128 meters, the maximum change that can be captured in 8 bits. Delta encoding over time is quite common in digital animation formats, since the change from one image frame to the next is also usually small. At the relatively low data resolutions for which IC was designed, however, the deltas between sample points (in both space and time) are too large for delta encoding to yield much data compression in the general case.

Another technique from image file formats segregates the bits of an integer pixel code into bitplanes and run-length encodes the bitplanes separately. Much of the compression benefit from this approach derives from its ability to encode different pixel depths (numbers of bits per pixel), which for IC would be impractical to implement. Note too that at 16 bits per pixel (or data point), the encoder must find 48 consecutive bits that are the same, aligned on 16-bit boundaries, before it can remove a single byte from the file size. Even for the highest order bits in typical IC density data—even for long runs of zeroes, in fact—this happens less frequently than one might suppose.

Other possibilities include various kinds of Huffman and Lempel-Ziv-Welch coding or a discrete cosine transform such as that used in JPEG (Joint Photographic Experts Group) image compression. The difficulty with these approaches, especially DCT, is their complexity, which has an impact both on program performance during decoding and on the ease with which programmers can implement their own IC file format code, and with the exception of DCT, dramatic improvement in compression is unlikely.

3.2.5 The IC File Format.

The binary format of simple data types in an IC data file is based on the internal formats of most popular C compilers running on Intel 80x86 PCs. The byte order places the most significant byte at the highest linear address, an order sometimes called big-endian in imitation of Swift (1727). As used here, the C types short and long are 2- and 4-byte 2's complement integers. The C type float is a 4-byte IEEE floating-point number (IEEE 1985). The header structure described below is packed, meaning that no extra bytes are allowed between structure members. The file format, however, aligns all data in the file on 2-byte boundaries.

IC data files consist of 3D density data preceded by a 118-byte header giving the size of the grid, the spacing of the data points along each axis, and other information about the data.

```
typedef struct {
   unsigned short     size;
   float              scale;
   float              offset;
} AXIST;

typedef struct {
   char               comment[ 40 ];
   char               reserved;
   unsigned short     percentile[ 9 ];
   long               creation_date;
   float              simulation_time;
   AXIST              axis[ 3 ];
} HEADER;
```

comment A 40-character NULL-terminated string identifying and describing the data. This string is displayed in the Level and Progress dialogs.

reserved This space is currently ignored.

percentile An array containing an approximate distribution for the densities. At each decile interval from 90% to 10%, the array gives the level at which that percentage of the non-zero data points will be enclosed in an isosurface at that level. This is the information displayed by the Level dialog. Levels are written in the same 16-bit format described in 3.2.3 for the density data.

creation_date This is a creation date and time written as a 4-byte integer in the manner of the ANSI function `time()`, which returns the number of seconds since 00:00:00 January 1, 1970. It is meant to document when the original data was created, but may also be used to record when the IC format file was generated from the data.

simulation_time If appropriate for the data, this stores the time in floating-point seconds from the time origin. For simulations that produce data at a sequence of time points, this field can be used to give the time coordinate. The simulation time is displayed in the Level and Progress dialogs.

axis An array of three structures giving the dimensions (**size**), distance between data points (**scale**), and **offsets** from the origin. The scale and offset values are in arbitrary units. Having a scale for each axis allows ICCanvas to anisotropically scale its surfaces for cases where the volumetric cells, or voxels, from which the data have presumably been sampled are not cubes. The offsets allow

ICCanvas to preserve spatial information in cases where a simulation has tracked a moving mass to keep it within the bounding box of its space. Both scale and offset are taken into account when a surface is saved with the Save Surface menu options.

The header is followed by the actual density data. Densities are expressed in arbitrary units, although by convention they have been in grams per cubic centimeter. Each density value is packed into 16 bits as described in 3.2.3. Densities from each x-y plane, or slice, of data are then run-length encoded as described in 3.2.2. Each compressed slice in the file is preceded by a 2-byte integer giving the compressed size of the slice.

An IC format data file may be summarized as

```
HEADER (118 bytes)
s1, size of first slice (2 bytes)
the first slice (s1 * 2 bytes)
   begins with n1, the first RLE code (2 bytes)
   first run
   RLE code n2
   second run...
s2 (2 bytes)
the second slice (s2 * 2 bytes)
   begins with n1, the first RLE code (2 bytes)
   ...
```

and so on for `header.axis[2].size` slices.

3.3 ADVANCED MICROCOMPUTER VISUALIZATION.

A part of our work on ICCanvas has been devoted to marrying the program's surface creation capability to sophisticated commercial 3D rendering applications available for small systems. The rendering software with which we have the most experience is called Lightwave. (There is some irony in the fact that Lightwave author Allen Hastings worked on cloud modeling for a defense contractor before turning to the development of commercial software.) Lightwave is used to produce all of the spaceflight scenes for the syndicated television show "Babylon 5" and many of the underwater scenes for the NBC series "Seaquest DSV."

The evolution of a nuclear cloud over time is captured by ICCanvas as a series of models, one for each time point in the density database. Ideally, each object would be loaded into the 3D rendering program at the appropriate frame on the basis of a sequence number that formed part of the object's filename. This is a frequently used technique for sequences of images. Consider a computer-animated scene in which a giant reptile threatens the patrons of a drive-in theater. The movie playing on the screen in the background is actually a planar image map. For each frame of the reptile animation, a single frame of the

movie is loaded and painted onto the screen object's surface. The association of each movie frame with a frame of reptile animation is automatic and is determined in part by sequence numbers imbedded in the filenames of the movie frames. Unfortunately, similar sequences of objects are not so readily supported, possibly because objects are presumed to be hand-made. There are other, more indirect approaches, however.

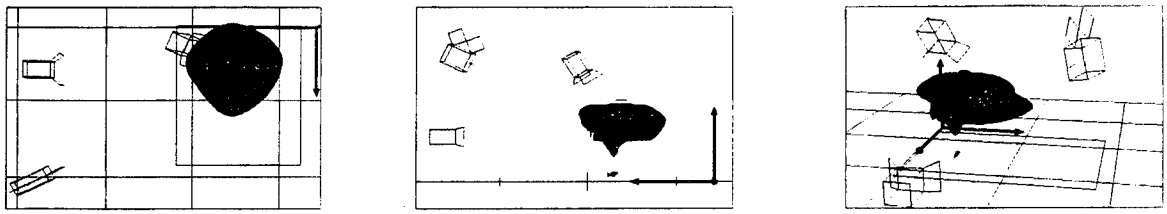
In Lightwave, the visibility of an object is controlled by a property called Dissolve. Dissolve is an object property distinct from Transparency, which is a surface property. An object all of whose surfaces are fully transparent may still participate in refraction and shadow calculations and may exhibit specular reflection (glossy highlights), whereas a fully dissolved object is considered absent from the scene. The loading of an object and the setting of its Dissolve property can be controlled from a script file, and it is a simple matter to write a program that automatically generates the appropriate script for any sequence of objects.

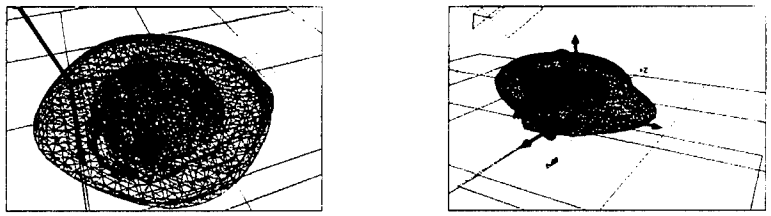
In the script fragment that follows, a step function is created for the Dissolve property that causes the object to be visible in a single frame of the animation.

```
LoadObject LW:Objects/tcw_0420.lw
ObjDissolve (envelope)
  1
  5
  1.0
  0 1 0.0 0.0 0.0
  1.0
  35 1 0.0 0.0 0.0
  0.0
  36 1 0.0 0.0 0.0
  1.0
  37 1 0.0 0.0 0.0
  1.0
  56 1 0.0 0.0 0.0
EndBehavior 1
ShadowOptions 7
```

Note that almost every time-dependent component of an animation, including Dissolve, is defined sparsely, through *keyframes*, and interpolated (*tweened*) for other frames. This fragment specifies 5 keyframes; the value of Dissolve is 0.0 at frame 36 and 1.0 everywhere else. A program is needed to generate these scripts because an animation involving even a few level surfaces per time point can require the precise specification of Dissolve envelopes for hundreds of objects.

As this example implies, the use of 3D rendering for visualization is different from the use of high-end software designed specifically for data analysis. 3D rendering is an artist's tool that has more in common with the making of a movie. In Figure 3-1, a single frame from a cloud animation is accompanied by 5 wireframe views of the "stage" on which the animation was created. A key light responsible for most of the illumination in the scene shines down on the cloud from above and creates a shadow on the ground. A lower intensity fill light brightens self-shadowed areas of the cloud. The semi-transparent green





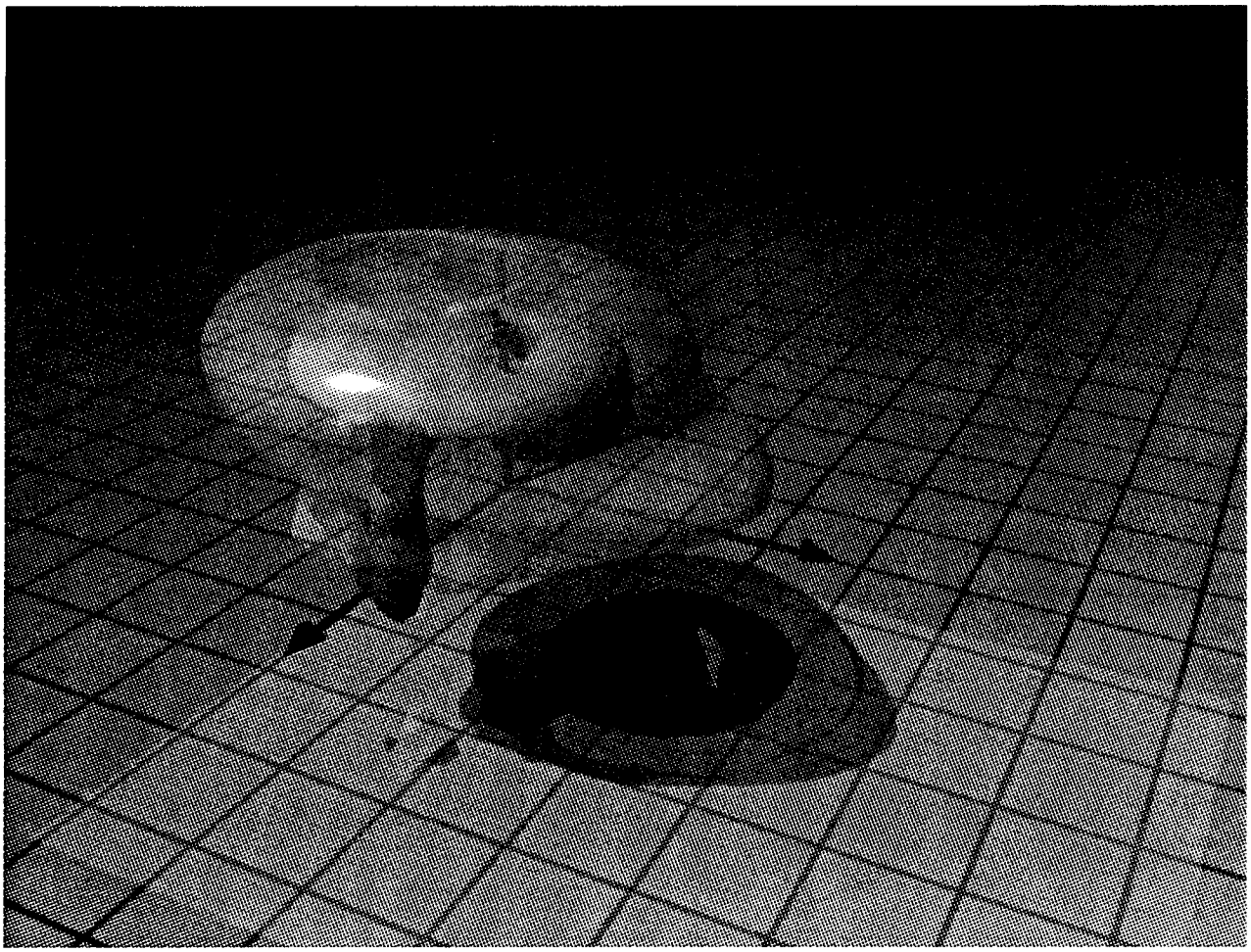


Figure 3-3. One frame from a cloud growth animation.
The wire diagrams show the scene from various points of view.

rectangle slides slowly across the ground in the animation to depict the way TASS shifted its origin to keep the cloud centered in its data space. The attenuation of the background brightness was created by an effect called Distant Fog; its purpose here is to focus attention on the objects in the foreground. The entire scene is anti-aliased to remove high frequency image components that in video cause artifacts such as stairstep lines, rainbows and jitter.

SECTION 4
REFERENCES

Lorenson, W., and H. Cline (1987). "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," *Computer Graphics.* **21**(4), 163-169.

Wyvill, G., C. McPheeters, and B. Wyvill (1986). "Data Structure for *Soft* Objects," *The Visual Computer.* **2**(4), 227-234.

Bloomenthal, J. (1988). "Polygonalisation of Implicit Surfaces," *Computer Aided Geometric Design.* **5**, 341-355.

Hall, M. (1990). "Defining Surfaces from Sampled Data," in A. Glassner, ed., *Graphics Gems.* Academic Press, Boston. 552-557.

Dürst, M. (1988). "Additional Reference to Marching Cubes," *Computer Graphics.* **22**(2), 72-73.

Gouraud, H. (1971). "Continuous Shading of Curved Surfaces," *IEEE Transactions on Computers.* **C-20**(6), 623-629.

Foley, J., A. van Dam, S. Feiner, and J. Hughes (1990). Computer Graphics: Principles and Practice. Addison-Wesley, Reading, MA. 1035-1036.

Wilhelms, J., and A. Van Gelder (1990). "Topological Considerations in Isosurface Generation," *Computer Graphics.* **21**(4), 163-169.

Moore, D., and J. Warren (1991). "Mesh Displacement: An Improved Contouring Method for Trivariate Data," Technical Report TR 91-166. Rice University, Department of Computer Science.

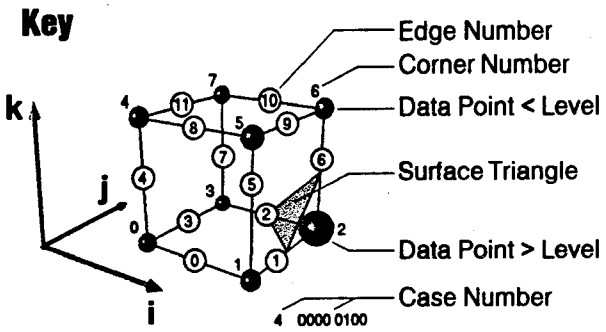
Moore, D., and J. Warren (1992).. "Compact Isocontours from Sampled Data," in D. Kirk, ed., *Graphics Gems III.* Academic Press, Boston. 23-28.

Hill, F. (1990). *Computer Graphics.* Macmillan, New York.

Heckbert, P. (1990). "Generic Convex Polygon Scan Conversion," in A. Glassner, ed., *Graphics Gems.* Academic Press, Boston. 84-86, 667-680.

IEEE (1985). "Standard for Binary Floating-Point Arithmetic," ANSI/IEEE 754-1985.

APPENDIX
THE CUBE CASES



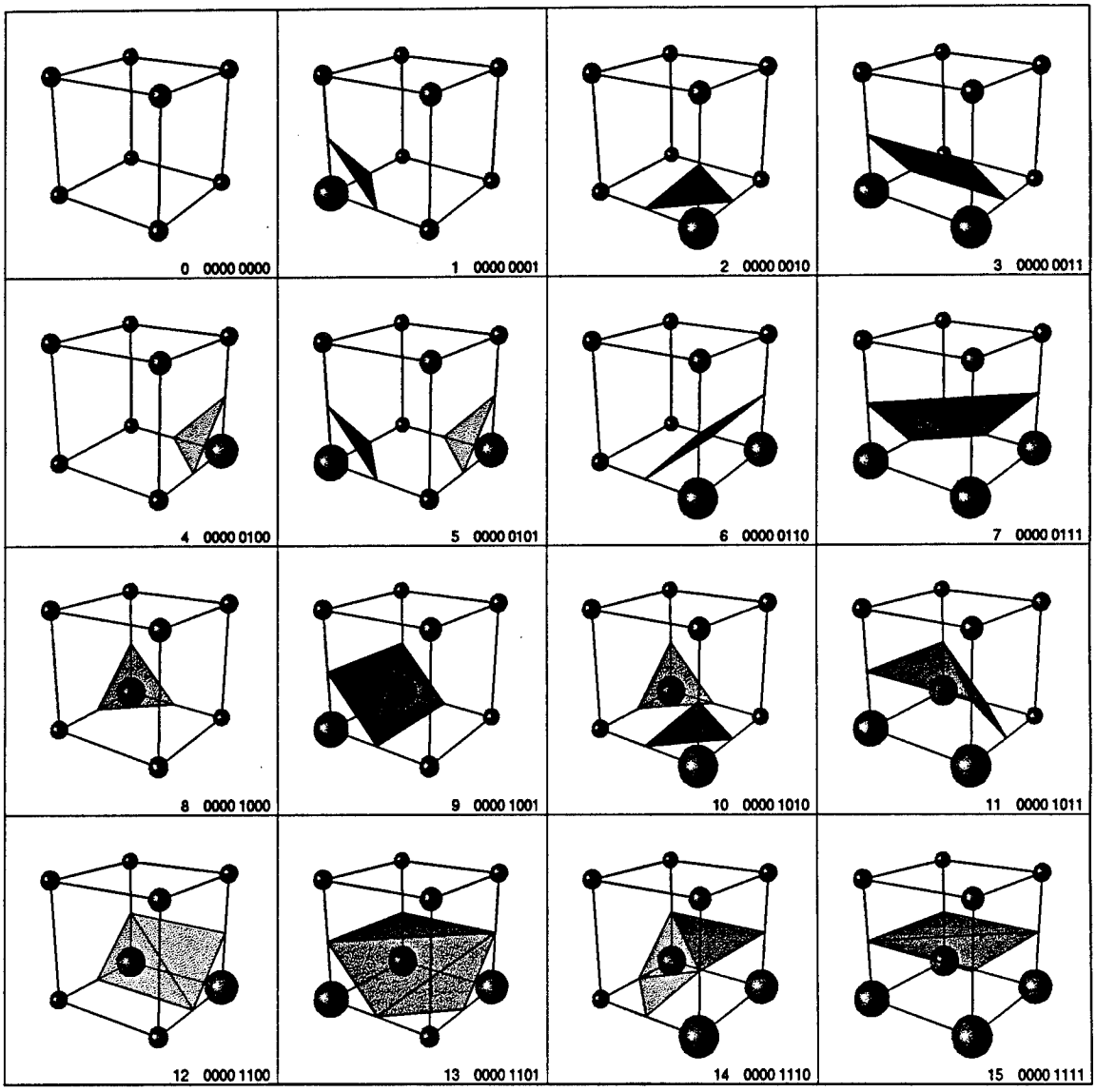


Figure A-1. The cube cases.

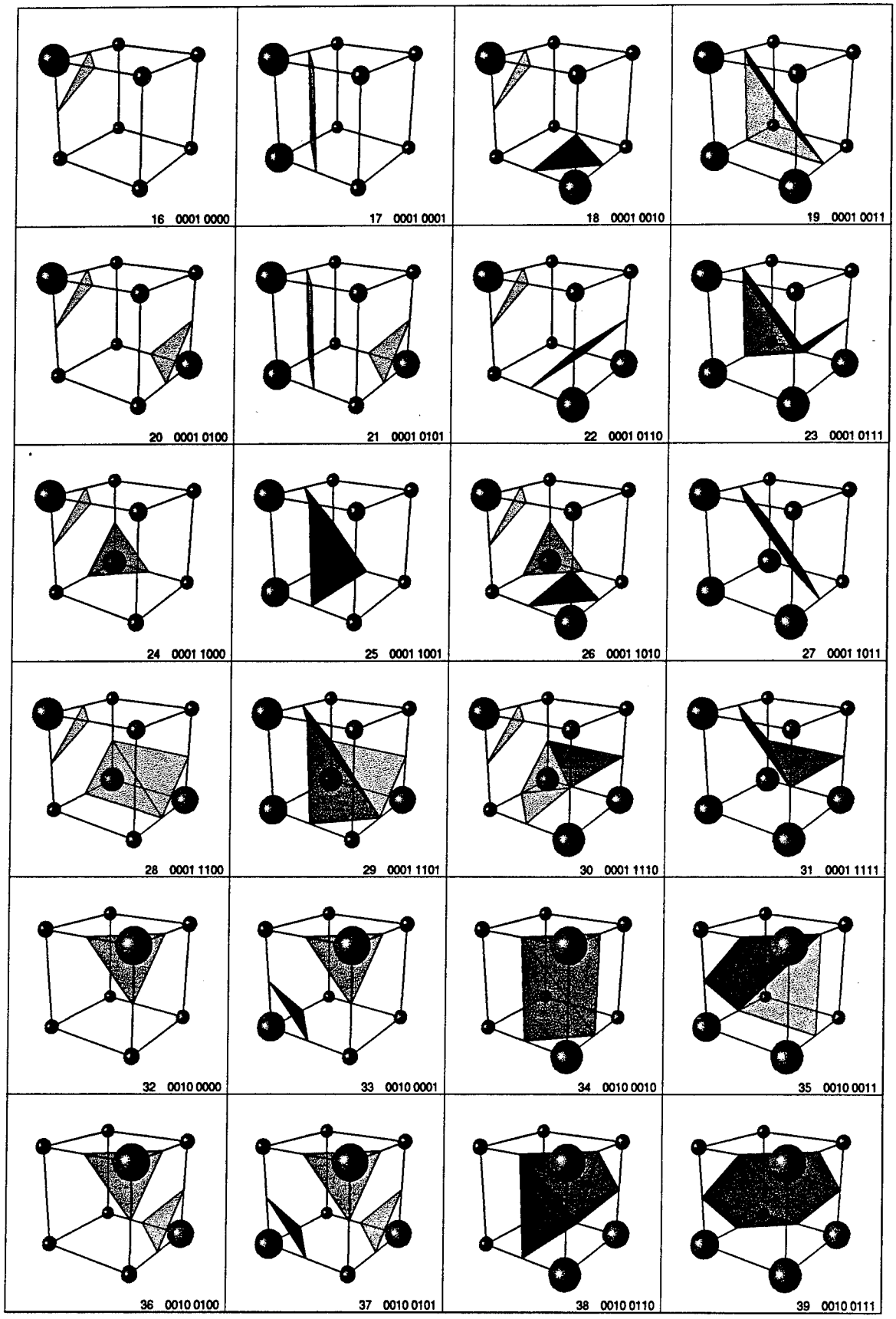


Figure A-1. The cube cases (continued).

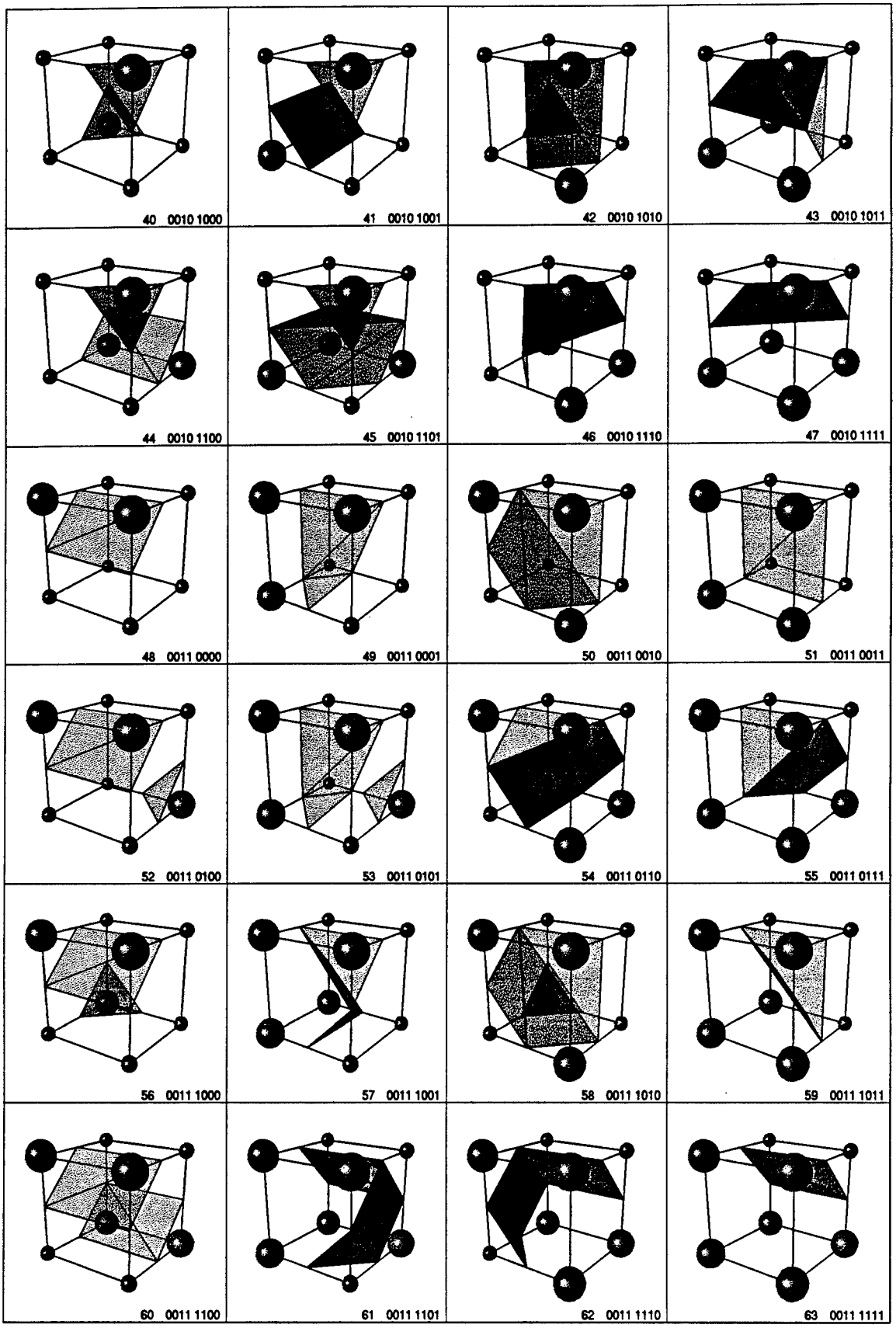


Figure A-1. The cube cases (continued).

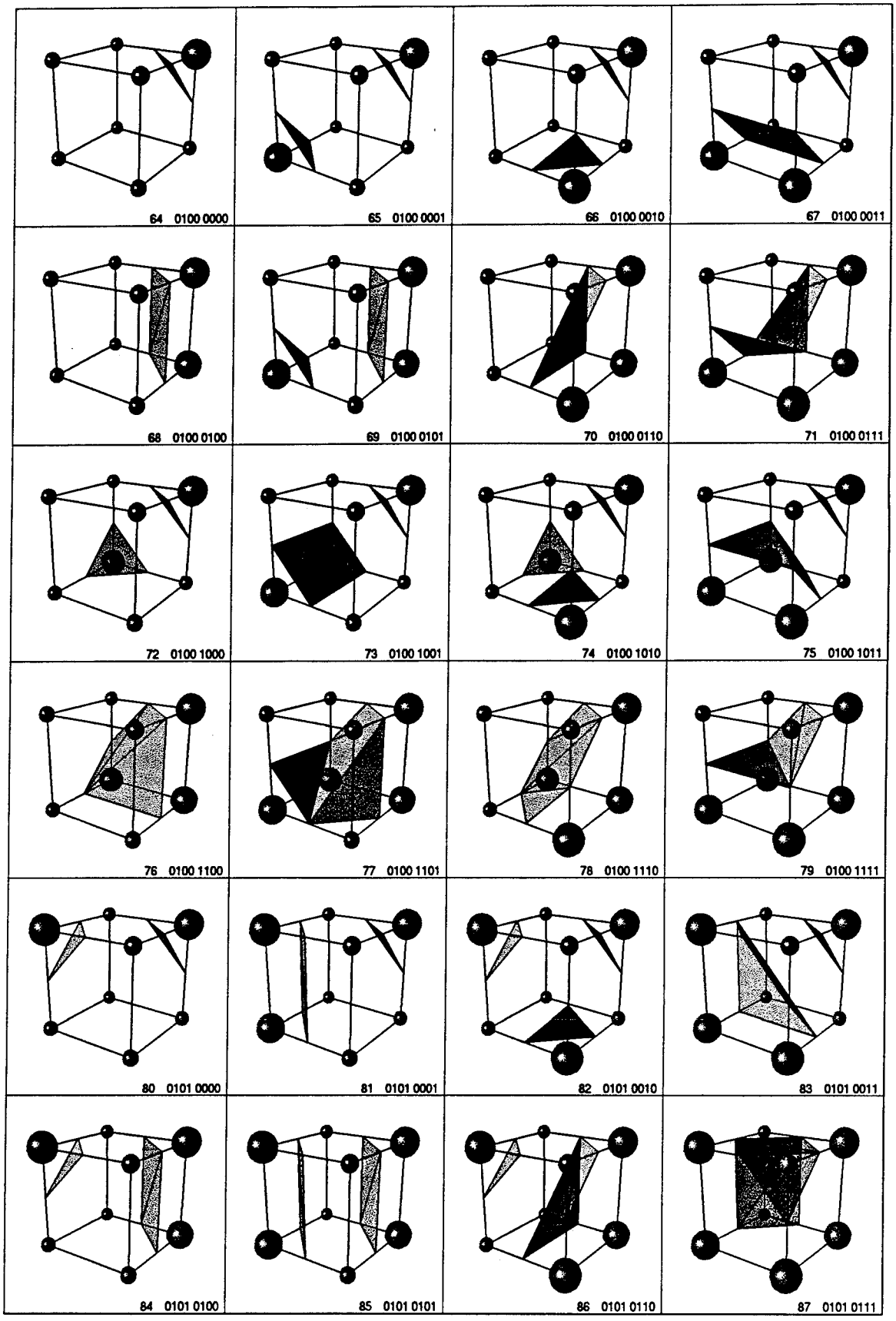


Figure A-1. The cube cases (continued).

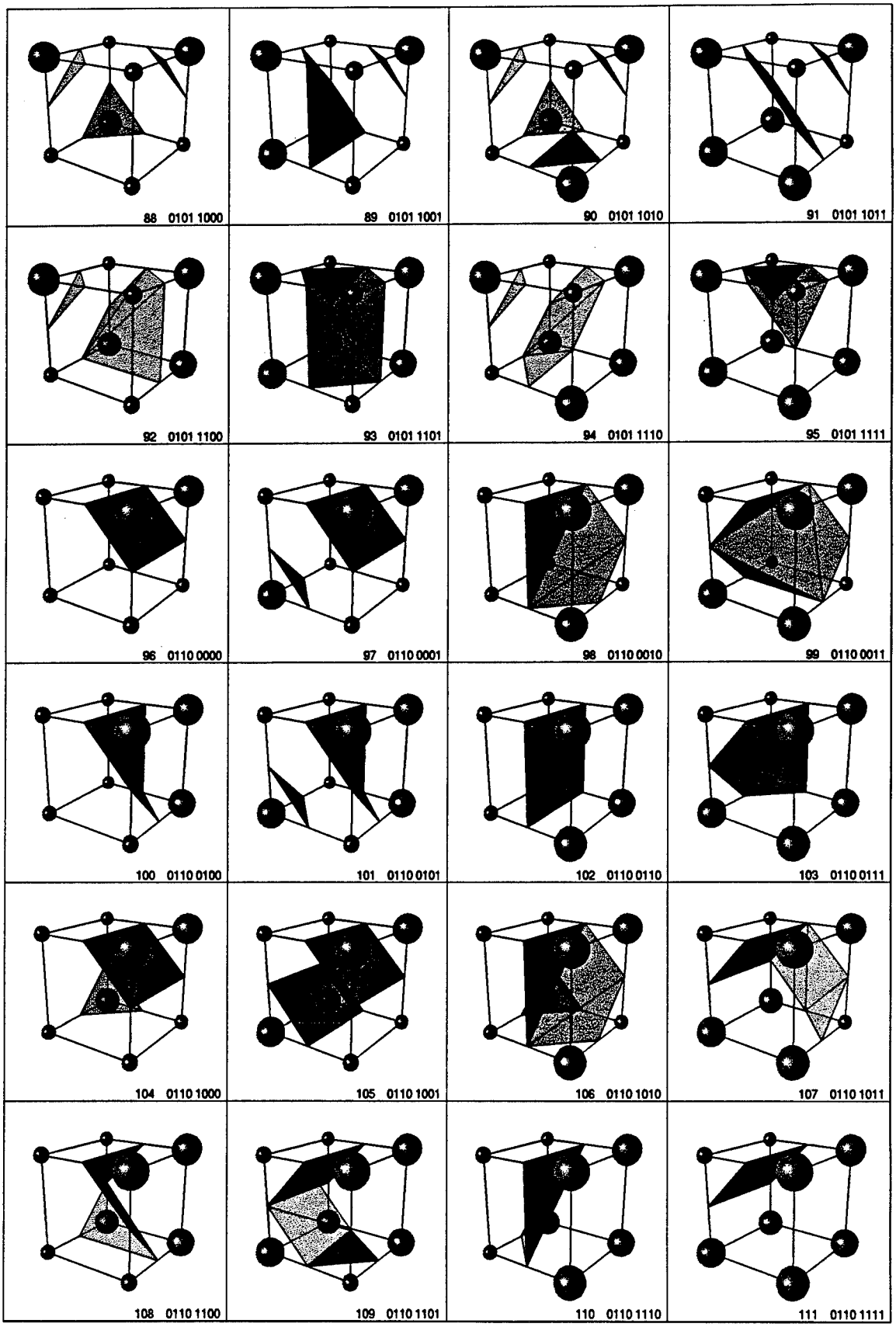


Figure A-1. The cube cases (continued).

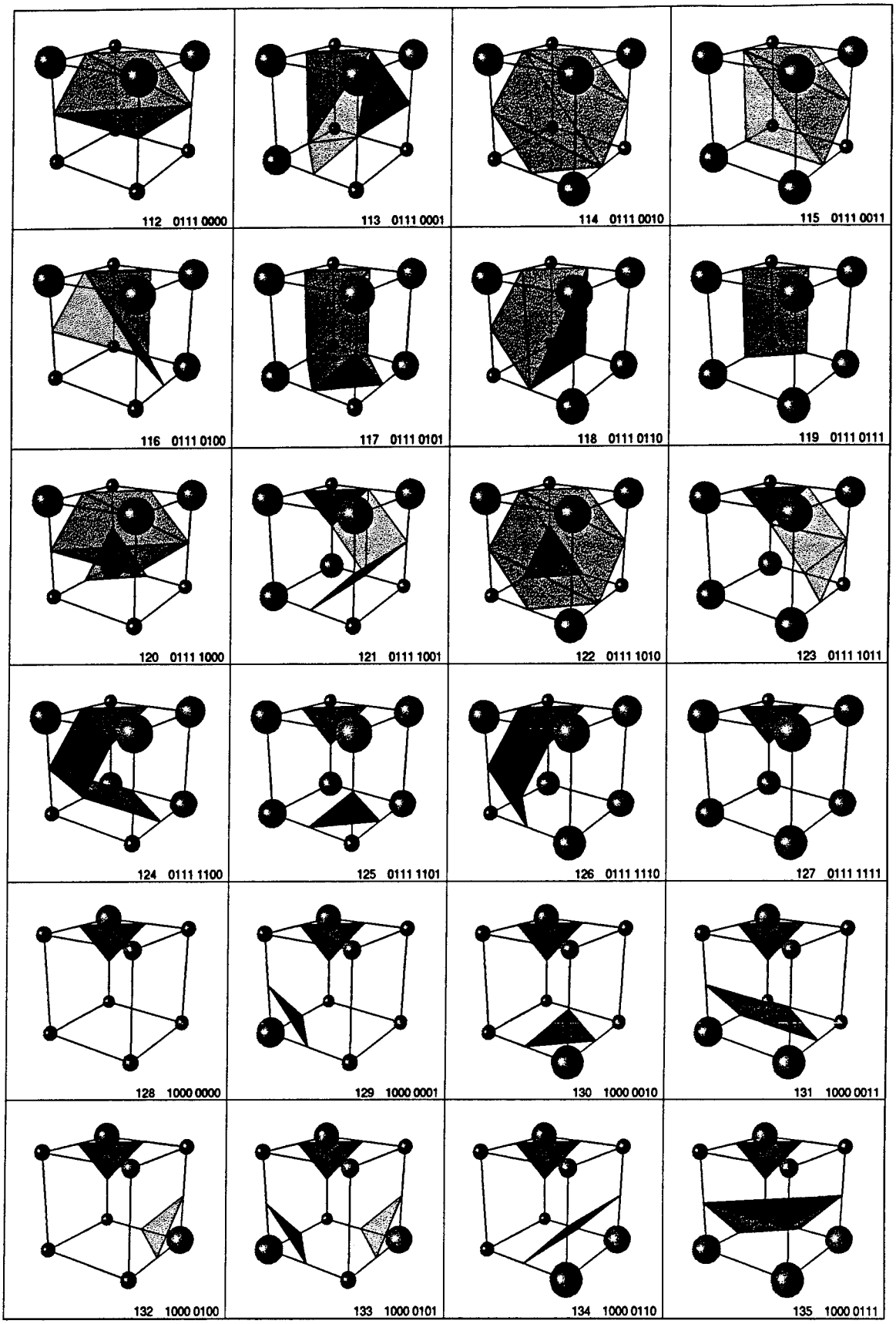


Figure A-1. The cube cases (continued).

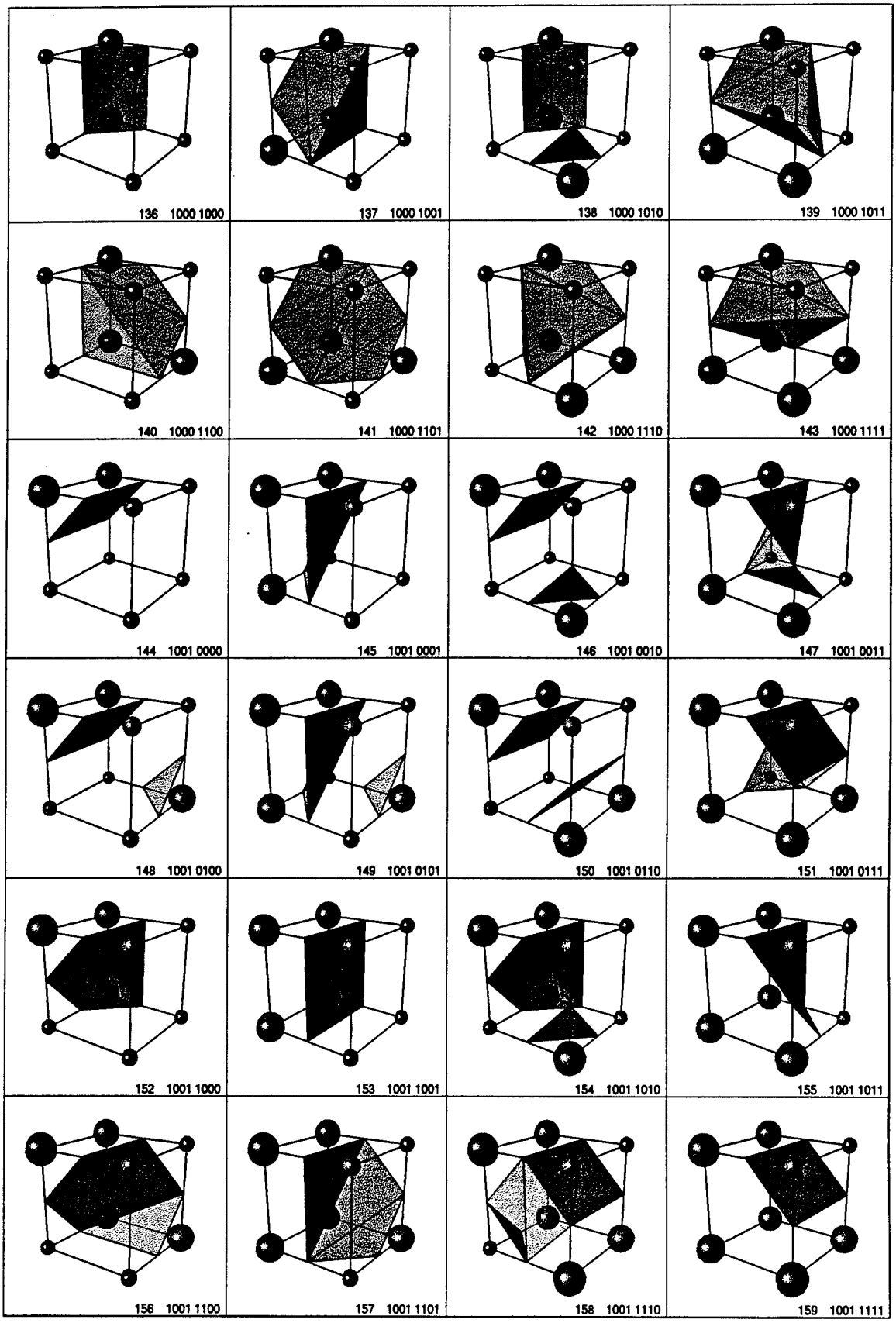


Figure A-1. The cube cases (continued).

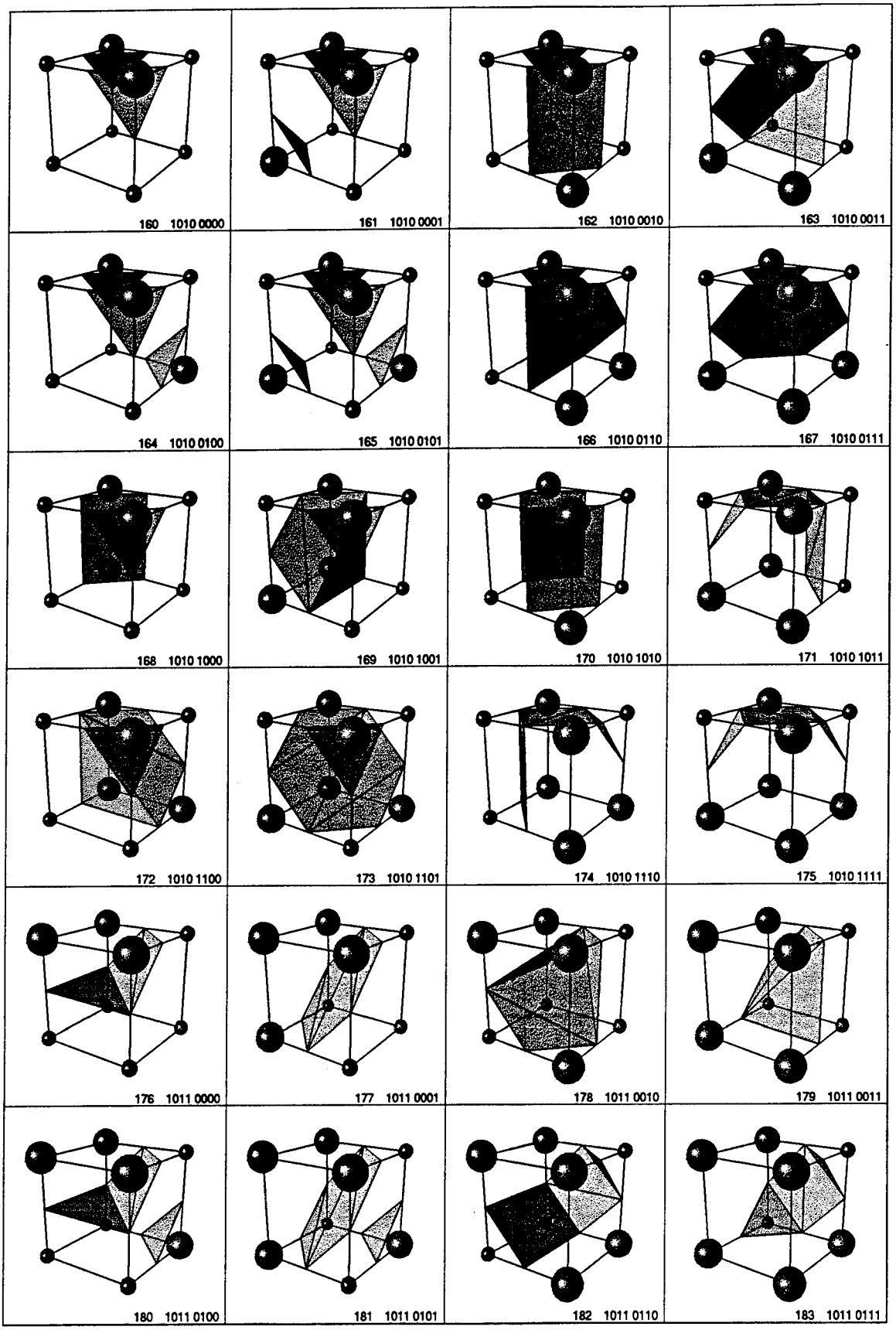


Figure A-1. The cube cases (continued).

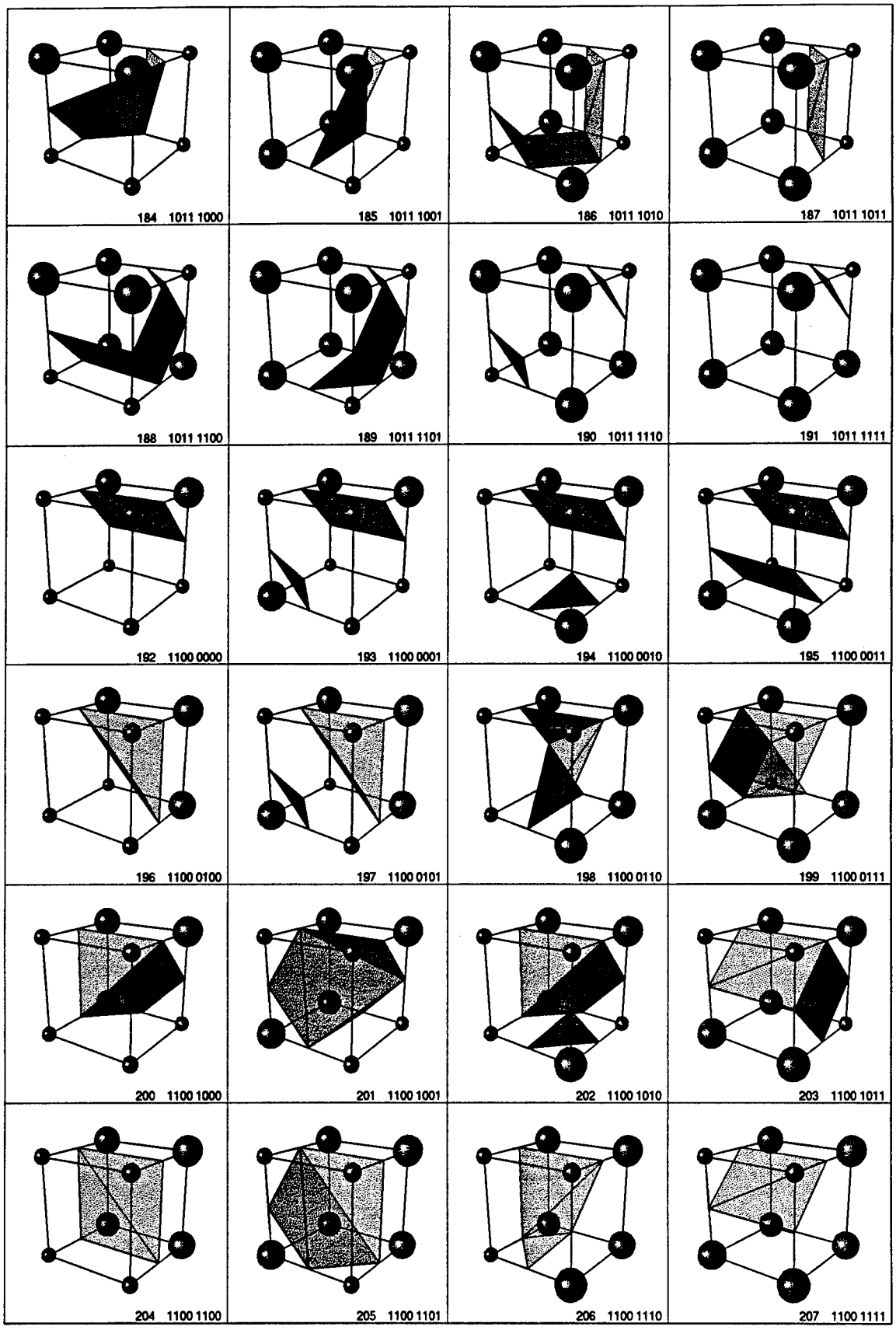


Figure A-1. The cube cases (continued).

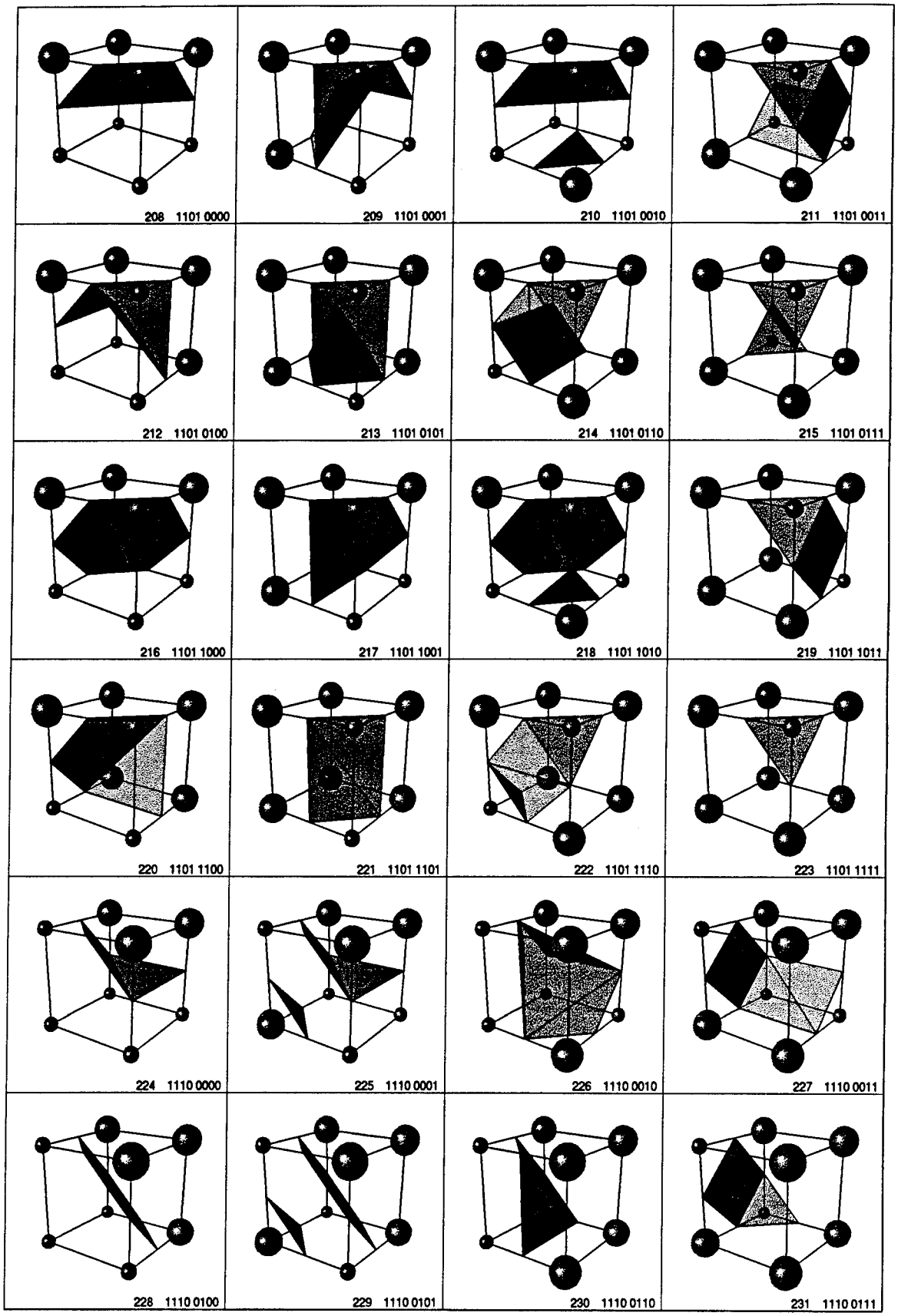


Figure A-1. The cube cases (continued).

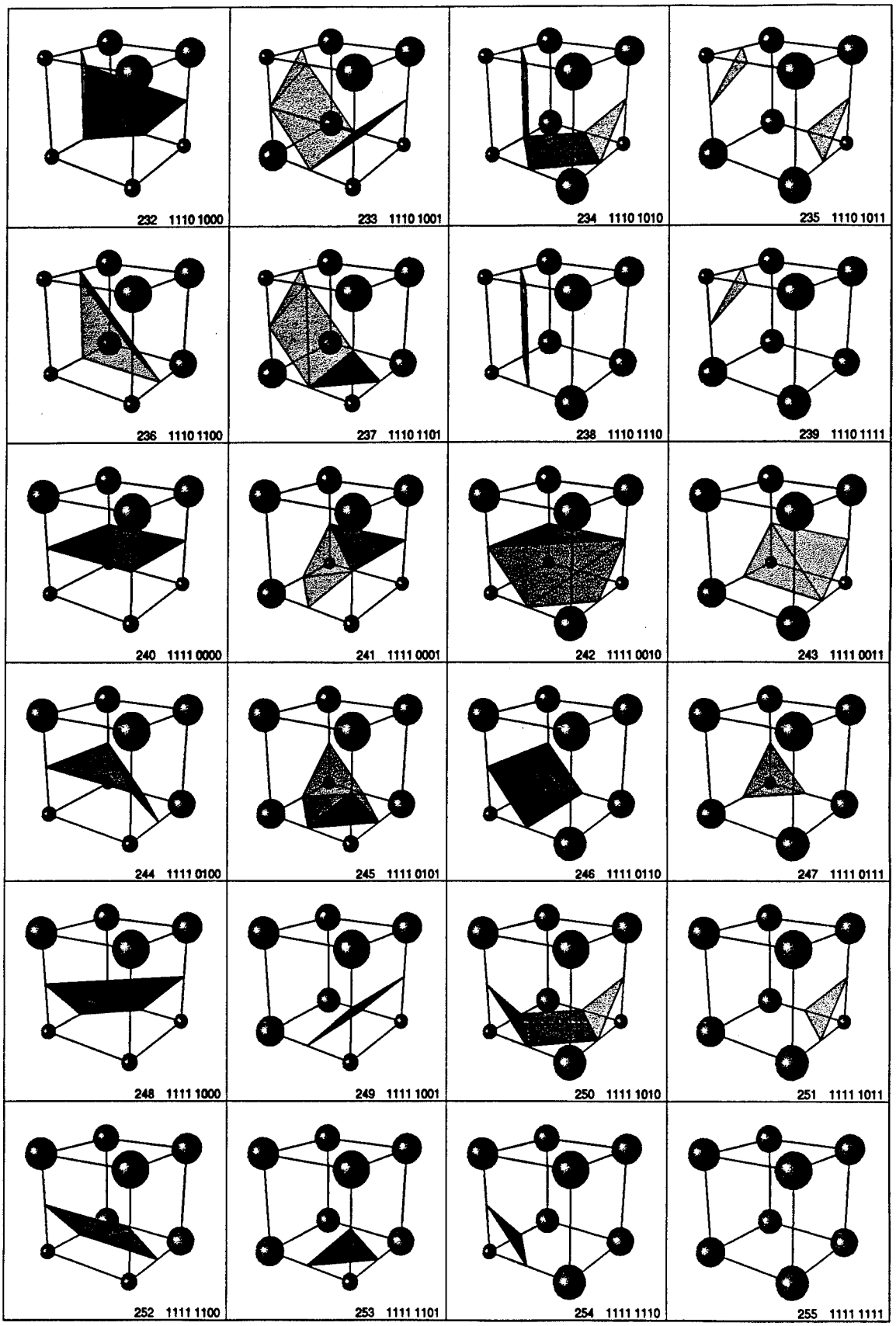


Figure A-1. The cube cases (continued).

DISTRIBUTION LIST

DNA-TR-94-64

DEPARTMENT OF DEFENSE

DEFENSE INTELLIGENCE AGENCY
ATTN: DGI4
ATTN: PAG1

DEFENSE NUCLEAR AGENCY
2 CY ATTN: IMTS
3 CY ATTN: OPNA MAJ SANDERS
ATTN: RAEM
ATTN: SPWE K PETERSEN
ATTN: SPWE LTC JIM HODGE
ATTN: SPWE LTC MARK BYERS

DEFENSE TECHNICAL INFORMATION CENTER
2 CY ATTN: DTIC/OC

FIELD COMMAND DEFENSE NUCLEAR AGENCY
2 CY ATTN: FCPR

DEPARTMENT OF THE ARMY

U S ARMY ATMOSPHERIC SCIENCES LAB
ATTN: SLCAS-AR-M

U S ARMY NUCLEAR & CHEMICAL AGENCY
ATTN: MONA-NU DR D BASH

U S ARMY TRAINING AND DOCTRINE COMD
ATTN: ATCD-N

US ARMY CHEMICAL SCHOOL
ATTN: ATZN-CM-CC-003

DEPARTMENT OF THE NAVY

NAVAL POSTGRADUATE SCHOOL
ATTN: CODE 52 LIBRARY

NAVAL SURFACE WARFARE CENTER
ATTN: CODE H-21

DEPARTMENT OF THE AIR FORCE

AIR FORCE INSTITUTE OF TECHNOLOGY/EN
ATTN: ENA

AIR UNIVERSITY LIBRARY
ATTN: AUL-LSE

ASSISTANT CHIEF OF STAFF
ATTN: AFSAA/SAK

DEPARTMENT OF ENERGY

LAWRENCE LIVERMORE NATIONAL LAB
ATTN: TECH LIBRARY

LOS ALAMOS NATIONAL LABORATORY
ATTN: TECH LIBRARY

SANDIA NATIONAL LABORATORIES
ATTN: TECH LIB 3141

DEPARTMENT OF DEFENSE CONTRACTORS

HORIZONS TECHNOLOGY, INC
ATTN: B LEE

JAYCOR
ATTN: CYRUS P KNOWLES

KAMAN SCIENCES CORP
ATTN: D MOFFETT
ATTN: DASIAC

KAMAN SCIENCES CORPORATION
ATTN: DASIAC

S-CUBED
ATTN: J NORTHROP

SCIENCE APPLICATIONS INTL CORP
ATTN: E SWICK

SCIENCE APPLICATIONS INTL CORP
ATTN: J MCGAHAN

VIRTUAL IMAGE LABS INC
2 CY ATTN: E WRIGHT